ECE 385

Spring 2021

Experiment 4

8-bit Multiplier

Wenhao Tan & Jiacheng Huang
ABI
Yanye Li

Introduction

In this lab, we are implementing an 8-bit multiplier using the add-shift algorithm. The multiplication process takes 8 steps. Each step the value is either added or subtracted and then shifted right by one bit to the register. After 8 cycles of add/shift, we have the result of the multiplication value stored into the two 8-bit registers.

Pre-Lab Question

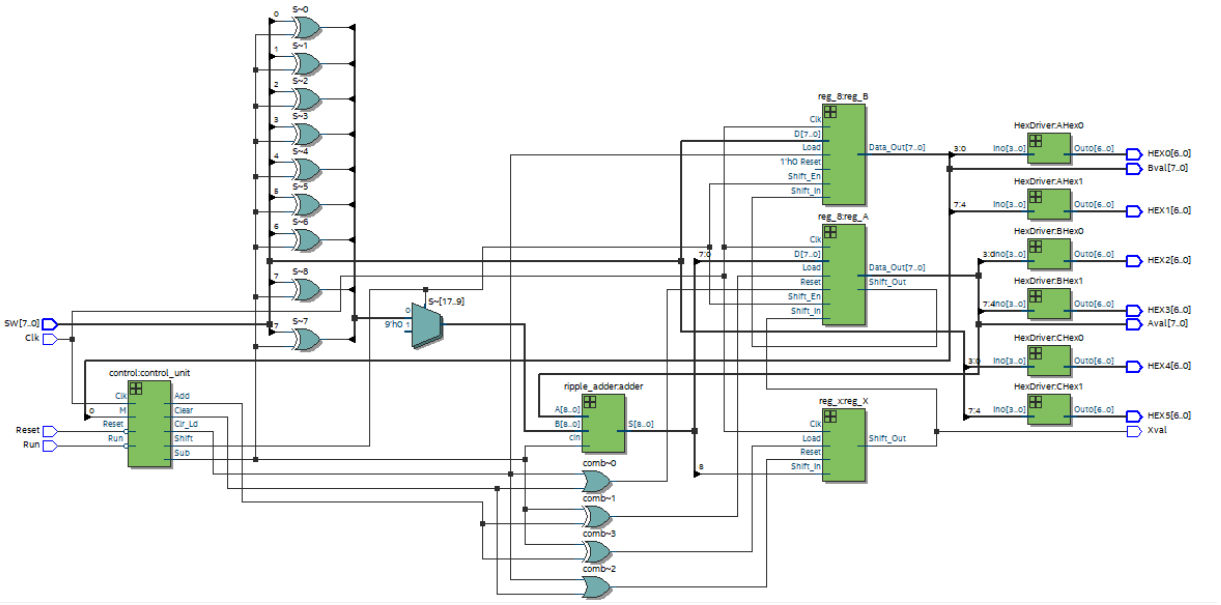
Function	X	A	B	M	Comments for next step
Clear A, Load B, Reset	0	00000000	00000111	1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	1	11000101	00000111	1	Shift XAB by one bit after ADD complete
SHIFT	1	11100010	10000011	1	Add S to A since M = 1.
ADD	1	10100111	10000011	1	Shift XAB by one bit after ADD complete
SHIFT	1	11010011	11000001	1	Add S to A since M = 1.
ADD	1	10011000	11000001	1	Shift XAB by one bit after ADD complete
SHIFT	1	11001100	01100000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11100110	00110000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11110011	00011000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11111001	10001100	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11111100	11000110	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11111110	01100011	1	8th shift done. Stop. 16-bit Product in AB.

Summary of Operation

The first value in the multiplication is stored in the switches SW[7:0]. The second value is stored in register B, which has 8 bits. Register A is initially reset to h'00. Register X is also reset to b'0. During the first 7 add/shift states, if the last bit of register B is 1, we add the value in the switches to register A using our adder. If the last bit of register B is 0, then we just add 0 to register A. We also update register X to be the same as the leading bit of register A. Then, we right shift register A and B; the shift in for register A is X, and shift in for register B is A[0]. During the last sub/shift state, if the last bit of register B is 1, then we need to perform a subtraction. To do so, we XOR the value in the switches with 1, which flips all the bits. We also

use 1 for the carry in bit to accommodate with the 2's complement requirement, and then we add this result to register A. If the last bit of register B is 0, then we just add 0 to register A. We update register X after the subtraction, and shift register A and B one last time the same way. After 8 shifts, the multiplication process is done, and the values in AB are our final result.

Top Level Block Diagram



Written Descriptions of .sv Modules

Module: `ripple_adder.sv`

Inputs: `[8:0] A, [8:0] B, cin`

Outputs: `[8:0] S, cout`

Description: This is a 9-bit Carry-Ripple Adder(CRA) created by combining 1 full-adder with two 4-bit carry-ripple adders, which is combined with four 1-bit full-adders. The adder performs bitwise addition from the lowest bit up one bit at a time.

Purpose: This module is used to calculate the resulting value of the new XA after adding old XA with 9-bit sign extended SW.

Module: `reg_8.sv`

Inputs: `[7:0] D, Clk, Reset, Shift_In, Load, Shift_En`

Outputs: `[7:0] Data_Out, Shift_Out`

Description: This is an 8-bit shift register created by combining two 4-bit shift registers. The register is positive-edge triggered with synchronous reset and load. When `Load` is high, data is loaded from `D` into the register on the positive edge of `Clk`.

Purpose: This module is used to store register A and B in the multiplier circuit.

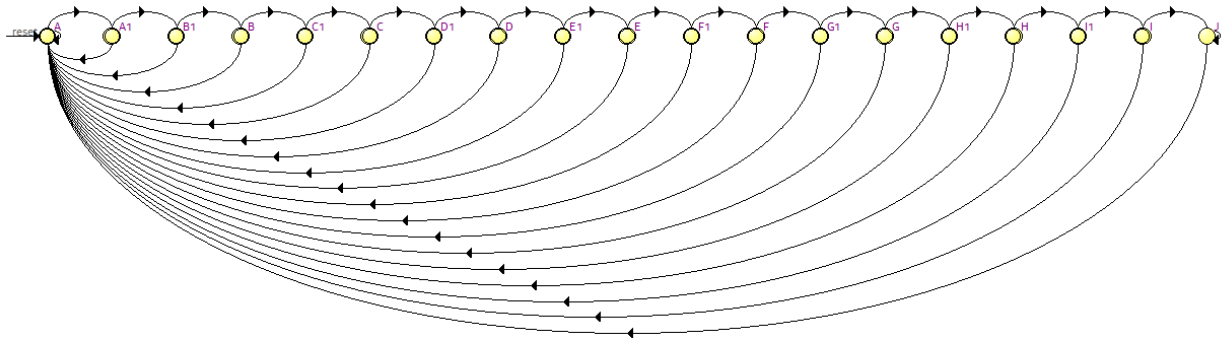
Module: `reg_x.sv`
Inputs: `Clk, Reset, Shift_In, Load`
Outputs: `[7:0] Data_Out, Shift_Out`
Description: This is a 1-bit flip-flop register. The register is positive-edge triggered with synchronous reset and load. When `Load` is high, data is loaded from `Shift_In` into the register on the positive edge of `Clk`.
Purpose: This module is used to store register X in the multiplier circuit.

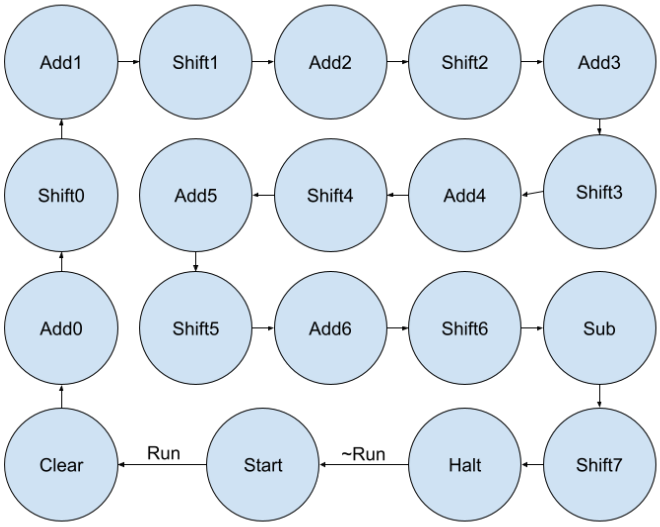
Module: `control.sv`
Inputs: `Clk, Reset, Run, M`
Outputs: `Clr_Ld, Clear, Shift, Add, Sub`
Description: This is the control module that keeps the state machine for the circuit. The state machine has a start/reset state A, a clear state A1, 8 ADD states (B1, C1, …, I1), 8 SHIFT states (B, C, …, I), and a halt state J. The state machine has synchronous reset and run, and all state transition happens on the positive edge of `Clk`. The default for all outputs are low. When `Reset` is high, the current state goes to start/reset state A, and `Clr_Ld` is high. When `Run` is high, the current state goes to clear state A1 from start/reset state A, and `Clear` is high. Then, the current state moves down the 8 ADD and SHIFT states alternatively. When in the first 7 ADD states, `Add` is `M`. `M` is bit 0 of register B, which is used to determine if we should add/subtract the value in SW or 0. When in SHIFT state, `Shift` is high. In the last ADD state, `Sub` is `M`. After the last shift state, the current state moves to halt state J, then back to start/reset state A if `Run` is low.
Purpose: This module is used to control the states that the circuit is in and decide what actions to perform during the current clock cycle.

Module: `adder2.sv`
Inputs: `[7:0] SW, Clk, Reset, Run`
Outputs: `[7:0] Aval, [7:0] Bval, Xval, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5`
Logics: `Load, Clear, Shift, Add, Sub, M, B_in, [8:0] S, [8:0] sum`
Description: This is the multiplier module, which is also the top entity, for the circuit. Register A, B, X, the adder, and the control unit are all created in this module. Register B is reset when `Reset` is high and is loaded when `Load` is high. Register A and X are reset when `Reset` or `Clear` are high, and are loaded when either `Add` or `Sub` are high. Register A and B shift right by 1 bit when `Shift` is high. The shift in for register A is X, and the shift in for register B is `B_in`, which is the shift out of register A. `M` is bit 0 of register B. `S` is the 9-bit sign extended value of `SW` XOR with `Sub`, because if `Sub` is 1, it means we are performing subtraction, then we need to flip the bits of `SW`. `sum` is the result of the 9-bit addition of `S` and XA. When performing subtraction, we need to use 2's complement and add 1 to the addition. So the Cin bit

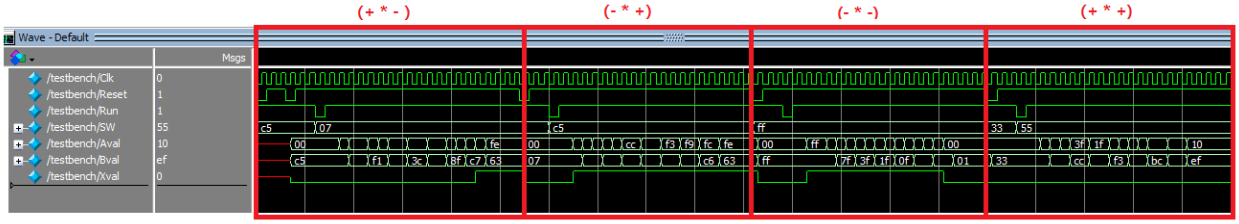
is `Sub`, which is 1 when we want to subtract. `Aval` and `Bval` are the values stored in register A and B, which are the result of the multiplication (the 16-bit value stored in AB).
Purpose: This module is used to perform the multiplication of the value in SW and register B, and storing the top 8 bits of the result in register A and bottom 8 bits of the result in register B.

State Diagram for Control Unit





Annotated Simulation Waveforms



00000111 * 11000101
= 11111110 01100011
(07 * C5 = FE63)

11000101 * 00000111
= 11111110 01100011
(C5 * 07 = FE63)

11111111 * 11111111
= 00000000 00000001
(FF * FF) = 0001

01010101 * 00110011
= 00010000 11101111
(55 * 33) = 10EF

Answers to The Post-Lab Questions

LUT	114
DSP	0
Memory (BRAM)	0
Flip-Flop	36
Frequency	64.75 MHz
Static Power	89.98 mW
Dynamic Power	1.72 mW
Total Power	107.17 mW

1. The design statistics table is shown above. In our design, we choose the carry ripple adder, because it is the most east-to-implement adder, but we may change it to a carry lookahead adder so that the efficiency will be increased a lot. Also, in our current design, there are 18 states, so we are considering adding a counter to reduce the amount of states and the amount of gates.

2. The X register helps us right shift A and B, so that the 17bits of XAB get updated properly. After each ADD, we set X to sum[8], the eighth bit of the results from the adder. In order to guarantee the proper behavior of consecutive multiplication, we clear X and A at the beginning of the next multiplication. If we don't have X and simply use a 8-bit adder instead of a 9-bit one, then we are not able to know the shift-in of the A register, and thus may cause the sign of the result to be incorrect. However, the design we just implemented still has limitations: if the result exceeds the range of [128, 127], then the continuous multiplication may fail, because the information is lost after X and A are cleared. This design is better than the pencil-and-paper method discussed in the introduction, because we only need three two 8-bit registers in order to perform the multiplication, but we may need 8 more to store the results of the multiplication of each bit in the pencil-and-paper method.

Conclusion

Our design is able to multiply two 8-bit 2's complement integers consecutively (multiply again immediately after the ending of the previous one) and produce a 16-bit output. During the lab, our ModelSim did not work, because we didn't reset the circuit manually in the test bench. After debugging, the circuit functions well now.

This lab manual provides detailed examples that show how a multiplier works step by step, which is perfect, but there are still some potential improvements to make. My suggestion is to add a separate section defining the registers that we use. For example, “B: 8-bit oprand and also the last 8 bits of the result; A: the first 8 bits of the result; S: the other 8-bit oprand, …”. In this way, we may feel more clear on the functions of each register. Otherwise, misunderstanding might be caused, since the naming “A” and “B” are more like operands during multiplication, and “S” is more like the “sum” of the two integers. We may still understand and memorize them after doing the lab for a while, but introducing them at the first glance is always a better idea.